SLIME User Manual

The Superior Lisp Interaction Mode for Emacs

1.2, $Date: 2005/04/18 18:58:12 $

Table of Contents

1 Introduction 1

2 Getting started 2
- 2.1 Supported Platforms 2
- 2.2 Downloading SLIME 2
  - 2.2.1 Downloading from CVS 2
  - 2.2.2 CVS incantations 2
- 2.3 Installation 3
- 2.4 Running SLIME 3

3 `slime-mode` 4
- 3.1 User-interface conventions 4
  - 3.1.1 Temporary buffers 4
  - 3.1.2 Key bindings 4
  - 3.1.3 `*inferior-lisp*` buffer 4
  - 3.1.4 Multithreading 5
- 3.2 Commands 5
  - 3.2.1 Compilation commands 5
  - 3.2.2 Finding definitions (“Meta-Point”). 6
  - 3.2.3 Lisp Evaluation 6
  - 3.2.4 Documentation 7
  - 3.2.5 Programming Helpers 7
  - 3.2.6 Abort/Recovery 8
  - 3.2.7 Cross-reference 8
  - 3.2.8 Inspector 9
  - 3.2.9 Profiling 10
- 3.3 Semantic indentation 10
- 3.4 Reader conditional fontification 10

4 REPL: the “top level” 12
- 4.1 REPL commands 12
- 4.2 Input navigation 12
- 4.3 Shortcuts 13

5 SLDB: the SLIME debugger 14
- 5.1 Examining frames 14
- 5.2 Invoking restarts 14
- 5.3 Navigating between frames 14
- 5.4 Miscellaneous Commands 15

6 Extras 16

6.1 `slime-selector` 16
6.2 `slime-autodoc-mode` 16
6.3 Multiple connections 16
6.4 Typeout frames 17

7 Customization 18

7.1 Emacs-side 18
7.1.1 Hooks 19
7.2 Lisp-side (Swank) 19
7.2.1 Communication style 19
7.2.2 Other configurables 20

8 Credits 22

Hackers of the good hack 22
Thanks! 22

1 Introduction

SLIME is the "Superior Lisp Interaction Mode for Emacs."

SLIME extends Emacs with new support for interactive programming in Common Lisp. The features are centred around `slime-mode`, an Emacs minor-mode that complements the standard `lisp-mode`. While `lisp-mode` supports editing Lisp source files, `slime-mode` adds support for interacting with a running Common Lisp process for compilation, debugging, documentation lookup, and so on.

The `slime-mode` programming environment follows the example of Emacs's native Emacs Lisp environment. We have also included good ideas from similar systems (such as ILISP) and some new ideas of our own.

SLIME is constructed from two parts: a user-interface written in Emacs Lisp, and a supporting server program written in Common Lisp. The two sides are connected together with a socket and communicate using an RPC-like protocol.

The Lisp server is primarily written in portable Common Lisp. The required implementation-specific functionality is specified by a well-defined interface and implemented separately for each Lisp implementation. This makes SLIME readily portable.

2 Getting started

This chapter tells you how to get SLIME up and running.

2.1 Supported Platforms

SLIME supports a wide range of operating systems and Lisp implementations. SLIME runs on Unix systems, Mac OSX, and Microsoft Windows. GNU Emacs versions 20 and 21 and XEmacs version 21 are supported.

The supported Lisp implementations, roughly ordered from the best-supported, are:

- CMU Common Lisp (CMUCL), 18e or newer
- Steel Bank Common Lisp (SBCL), from version 0.8.15 to 0.8.21 (newer versions may or may not work)
- OpenMCL, version 0.14.3
- LispWorks, version 4.3 or newer
- Allegro Common Lisp (ACL), version 6 or newer
- CLISP, version 2.33.2 or newer
- Armed Bear Common Lisp (ABCL)

Most features work uniformly across implementations, but some are prone to variation. These include the precision of placing compiler-note annotations, XREF support, and fancy debugger commands (like "restart frame").

2.2 Downloading SLIME

You can choose between using a released version of SLIME or accessing our CVS repository directly. You can download the latest released version from our website:

`http://www.common-lisp.net/project/slime/`

We recommend that users who participate in the `slime-devel` mailing list use the CVS version of the code.

2.2.1 Downloading from CVS

SLIME is available from the CVS repository on '`common-lisp.net`'. You have the option to use either the very latest code or the tagged `FAIRLY-STABLE` snapshot.

The latest version tends to have more features and fewer bugs than the `FAIRLY-STABLE` version, but it can be unstable during times of major surgery. As a rule-of-thumb recommendation we suggest that if you follow the `slime-devel` mailing list then you're better off with the latest version (we'll send a note when it's undergoing major hacking). If you don't follow the mailing list you won't know the status of the latest code, so tracking `FAIRLY-STABLE` or using a released version is the safe option.

If you checkout from CVS then remember to `cvs update` occasionally. Improvements are continually being committed, and the `FAIRLY-STABLE` tag is moved forward from time to time (about once per month).

2.2.2 CVS incantations

To download SLIME you first configure your `CVSROOT` and login to the repository.

```
export CVSROOT=:pserver:anonymous@common-lisp.net:/project/slime/cvsroot
cvs login
```

(The password is `anonymous`*)*

The latest version can then be checked out with:

```
cvs checkout slime
```

Or the `FAIRLY-STABLE` version can be checked out with:

```
cvs checkout -rFAIRLY-STABLE slime
```

If you want to find out what's new since the version you're currently running, you can diff the local '`ChangeLog`' against the repository version:

```
cvs diff -rHEAD ChangeLog      # or: -rFAIRLY-STABLE
```

2.3 Installation

With a Lisp implementation that can be started from the command-line, installation just requires a few lines in your '`~/.emacs`':

```
(setq inferior-lisp-program "the path to your Lisp system")
(add-to-list 'load-path "the path of your 'slime' directory")
(require 'slime)
(slime-setup)
```

The snippet above also appears in the '`README`' file. You can copy&paste it from there, but remember to fill in the appropriate path.

We recommend not loading the ILISP package into Emacs if you intend to use SLIME. Doing so will add a lot of extra bindings to the keymap for Lisp source files that may be confusing and may not work correctly for a Lisp process started by SLIME.

2.4 Running SLIME

SLIME is started with the Emacs command *`M-x slime`*. This uses the `inferior-lisp` package to start a Lisp process, loads and starts the Lisp-side server (known as "Swank"), and establishes a socket connection between Emacs and Lisp. Finally a REPL buffer is created where you can enter Lisp expressions for evaluation.

At this point SLIME is up and running and you can start exploring.

3 slime-mode

SLIME's commands are provided via `slime-mode`, a minor-mode used in conjunction with Emacs's `lisp-mode`. This chapter describes the `slime-mode` and its relatives.

3.1 User-interface conventions

To use SLIME comfortably it is important to understand a few "global" user-interface characteristics. The most important principles are described in this section.

3.1.1 Temporary buffers

Some SLIME commands create temporary buffers to display their results. Although these buffers usually have their own special-purpose major-modes, certain conventions are observed throughout.

Temporary buffers can be dismissed by pressing `q`. This kills the buffer and restores the window configuration as it was before the buffer was displayed. Temporary buffers can also be killed with the usual commands like `kill-buffer`, in which case the previous window configuration won't be restored.

Pressing *RET* is supposed to "do the most obvious useful thing." For instance, in an apropos buffer this prints a full description of the symbol at point, and in an XREF buffer it displays the source code for the reference at point. This convention is inherited from Emacs's own buffers for apropos listings, compilation results, etc.

Temporary buffers containing Lisp symbols use `slime-mode` in addition to any special mode of their own. This makes the usual SLIME commands available for describing symbols, looking up function definitions, and so on.

3.1.2 Key bindings

In general we try to make our key bindings fit with the overall Emacs style. We also have the following somewhat unusual convention of our own: when entering a three-key sequence, the final key can be pressed either with control or unmodified. For example, the `slime-describe-symbol` command is bound to *C-c C-d d*, but it also works to type *C-c C-d C-d*. We're simply binding both key sequences because some people like to hold control for all three keys and others don't, and with the two-key prefix we're not afraid of running out of keys.

There is one exception to this rule, just to trip you up. We never bind *C-h* anywhere in a key sequence, so *C-c C-d C-h* doesn't do the same thing as *C-c C-d h*. This is because Emacs has a builtin default so that typing a prefix followed by *C-h* will display all bindings starting with that prefix, so *C-c C-d C-h* will actually list the bindings for all documentation commands. This feature is just a bit too useful to clobber!

3.1.3 *inferior-lisp* buffer

SLIME internally uses the inferior-lisp package to start Lisp processes. This has a few user-visible consequences, some good and some not-so-terribly. To avoid confusion it is useful to understand the interactions.

The buffer *inferior-lisp* contains the Lisp process's own top-level. This direct access to Lisp is useful for troubleshooting, and some degree of SLIME integration is available using the inferior-slime-mode. However, in normal use we recommend using the fully-integrated SLIME REPL and ignoring the *inferior-lisp* buffer.

An unfortunate property of inferior-lisp is it inserts some commands of its own directly into the lisp-mode keymap, such that they aren't easily disabled. This makes Lisp source buffers slightly schizophrenic, having both SLIME and inferior-lisp commands bound to keys and operating independently.

SLIME overrides most key bindings, so in practice you are unlikely to accidentally use an inferior-lisp command. If you do find a command that pops up the *inferior-lisp* buffer, that command doesn't belong to SLIME, and you should probably lookup our equivalent.

3.1.4 Multithreading

If the Lisp system supports multithreading, SLIME spawns a new thread for each request, e.g., *C-x C-e* creates a new thread to evaluate the expression. An exception to this rule are requests from the REPL: all commands entered in the REPL buffer are evaluated in a dedicated REPL thread.

Some complications arise with multithreading and special variables. Non-global special bindings are thread-local, e.g., changing the value of a let bound special variable in one thread has no effect on the binding of the variables with the same name in other threads. This makes it sometimes difficult to change the printer or reader behaviour for new threads. The variable swank:*default-worker-thread-bindings* was introduced for such situtuations: instead of modifying the global value of a variable, add a binding the swank:*default-worker-thread-bindings*. E.g., with the following code, new threads will read floating point values as doubles by default:

```
(push '(*read-default-float-format* . double-float)
      swank:*default-worker-thread-bindings*).
```

3.2 Commands

3.2.1 Compilation commands

SLIME has fancy commands for compiling functions, files, and packages. The fancy part is that notes and warnings offered by the Lisp compiler are intercepted and annotated directly onto the corresponding expressions in the Lisp source buffer. (Give it a try to see what this means.)

C-c C-k slime-compile-and-load-file
Compile and load the current buffer's source file.

C-c M-k slime-compile-file
Compile (but don't load) the current buffer's source file.

C-c C-c slime-compile-defun
Compile the top-level form at point.

The annotations are indicated as underlining on source forms. The compiler message associated with an annotation can be read either by placing the mouse over the text or with the selection commands below.

M-n
M-p slime-next-note, slime-previous-note
These commands move the point between compiler notes and display the new note.

C-c M-c slime-remove-notes
Remove all annotations from the buffer.

3.2.2 Finding definitions ("Meta-Point").

The familiar *M-.* command is provided. For generic functions this command finds all methods, and with some systems it does other fancy things (like tracing structure accessors to their DEFSTRUCT definition).

M-. slime-edit-definition
Go to the definition of the symbol at point.

M-, slime-pop-find-definition-stack Go back from a definition found with *M-.*. This gives multi-level backtracking when *M-.* has been used several times.

3.2.3 Lisp Evaluation

These commands each evaluate a Lisp expression in a different way. By default they show their results in a message, but a prefix argument causes the results to be printed in the REPL instead.

C-M-x slime-eval-defun
Evaluate top-level form.

C-x C-e slime-eval-last-expression
Evaluate the expression before point.

C-c C-p slime-pprint-eval-last-expression
Evaluate the expression before point and pretty-print the result.

C-c C-r slime-eval-region
Evaluate the region.

C-c : slime-interactive-eval
Evaluate an expression read from the minibuffer.

M-x slime-scratch
: Create a '*slime-scratch*' buffer. In this buffer you can enter Lisp expressions and evaluate them with *C-j*, like in Emacs's '*scratch*' buffer.

If 'C-M-x' or 'C-x C-e' is given a numeric argument, it inserts the value into the current buffer at point, rather than displaying it in the echo area.

3.2.4 Documentation

SLIME's online documentation commands follow the example of Emacs Lisp. The commands all share the common prefix *C-c C-d* and allow the final key to be modified or unmodified (See Section 3.1.2 [Key bindings], page 4.)

C-c C-d d slime-describe-symbol
: Describe the symbol at point.

C-c C-d a slime-apropos
: Apropos search. Search Lisp symbol names for a substring match and present their documentation strings. By default the external symbols of all packages are searched. With a prefix argument you can choose a specific package and whether to include unexported symbols.

C-c C-d z slime-apropos-all
: Like slime-apropos but also includes internal symbols by default.

C-c C-d p slime-apropos-package
: Show apropos results of all symbols in a package. This command is for browsing a package at a high-level. With package-name completion it also serves as a rudimentary Smalltalk-ish image-browser.

C-c C-d h slime-hyperspec-lookup
: Lookup the symbol at point in the *Common Lisp Hyperspec*. This uses the familiar 'hyperspec.el' to show the appropriate section in a web browser. The Hyperspec is found either on the Web or in common-lisp-hyperspec-root, and the browser is selected by browse-url-browser-function.

C-c C-d ~ common-lisp-hyperspec-format
: Lookup a format character in the *Common Lisp Hyperspec*.

3.2.5 Programming Helpers

M-TAB slime-complete-symbol
: Complete the symbol at point. Note that three styles of completion are available in SLIME, and the default differs from normal Emacs completion. See Section 7.1 [Emacs-side customization], page 18.

C-c M-i slime-fuzzy-complete-symbol
: Presents a list of likely completions to choose from for an abbreviation at point. This is a third completion method and it is very different from the more traditional completion to which slime-complete-symbol defaults. It attempts to complete a symbol all at once, instead of in pieces. For example, "mvb"

will find "multiple-value-bind" and "norm-df" will find "least-positive-normalized-double-float". This can also be selected as the method of completion used for slime-complete-symbol.

SPC slime-space
The space key inserts a space and also looks up and displays the argument list for the function at point, if there is one.

C-c C-s slime-insert-arglist
Looks up and inserts into the current buffer the argument list for the function at point, if there is one.

C-c C-m slime-macroexpand-1
Macroexpand the expression at point once. If invoked with a prefix argument, use macroexpand instead of macroexpand-1.

C-c M-m slime-macroexpand-all
Fully macroexpand the expression at point.

C-c C-t slime-toggle-trace-fdefinition
Toggle tracing of the function at point. If invoked with a prefix argument, read additional information, like which particular method should be traced.

C-c M-d slime-disassemble-symbol
Disassemble the function definition of the symbol at point.

3.2.6 Abort/Recovery

C-c C-b slime-interrupt
Interrupt Lisp (send SIGINT).

C-c ~ slime-sync-package-and-default-directory
Synchronize the current package and working directory from Emacs to Lisp.

C-c M-p slime-repl-set-package
Set the current package of the REPL.

3.2.7 Cross-reference

SLIME's cross-reference commands are based on the support provided by the Lisp system, which varies widely between Lisps. For systems with no builtin XREF support SLIME queries a portable XREF package, which is taken from the *CMU AI Repository* and bundled with SLIME.

Each command operates on the symbol at point, or prompts if there is none. With a prefix argument they always prompt. You can either enter the key bindings as shown here or with the control modified on the last key, See Section 3.1.2 [Key bindings], page 4.

C-c C-w c slime-who-calls
Show function callers.

C-c C-w r slime-who-references
Show references to global variable.

C-c C-w b slime-who-binds
Show bindings of a global variable.

C-c C-w s slime-who-sets
Show assignments to a global variable.

C-c C-w m slime-who-macroexpands
Show expansions of a macro.

M-x slime-who-specializes
Show all known methods specialized on a class.

There are also "List callers/callees" commands. These operate by rummaging through function objects on the heap at a low-level to discover the call graph. They are only available with some Lisp systems, and are most useful as a fallback when precise XREF information is unavailable.

C-c < slime-list-callers
List callers of a function.

C-c > slime-list-callees
List callees of a function.

3.2.8 Inspector

The SLIME inspector is a very fancy Emacs-based alternative to the standard INSPECT function. The inspector presents objects in Emacs buffers using a combination of plain text, hyperlinks to related objects, and "actions" that can be selected to invoke Lisp code on the inspected object. For example, to present a generic function the inspector shows the documentation in plain text and presents each method with both a hyperlink to inspect the method object and a "remove method" action that you can invoke interactively.

The inspector can easily be specialized for the objects in your own programs. For details see the the inspect-for-emacs generic function in 'swank-backend.lisp'.

C-c I slime-inspect
Inspect the value of an expression entered in the minibuffer.

The standard commands available in the inspector are:

RET slime-inspector-operate-on-point
If point is on a value then recursivly call the inspcetor on that value. If point is on an action then call that action.

d slime-inspector-describe
Describe the slot at point.

l slime-inspector-pop
Go back to the previous object (return from *RET*).

n slime-inspector-next
The inverse of *l*. Also bound to *SPC*.

q slime-inspector-quit
Dismiss the inspector buffer.

M-RET `slime-inspector-copy-down`
Evaluate the value under point via the REPL (to set '*').

3.2.9 Profiling

M-x slime-toggle-profile-fdefinition
Toggle profiling of a function.

M-x slime-profile-package
Profile all functions in a package.

M-x slime-unprofile-all
Unprofile all functions.

M-x slime-profile-report
Report profiler data.

M-x slime-profile-reset
Reset profiler data.

3.3 Semantic indentation

SLIME automatically discovers how to indent the macros in your Lisp system. To do this the Lisp side scans all the macros in the system and reports to Emacs all the ones with `&body` arguments. Emacs then indents these specially, putting the first arguments four spaces in and the "body" arguments just two spaces, as usual.

This should "just work." If you are a lucky sort of person you needn't read the rest of this section.

To simplify the implementation, SLIME doesn't distinguish between macros with the same symbol-name but different packages. This makes it fit nicely with Emacs's indentation code. However, if you do have several macros with the same symbol-name then they will all be indented the same way, arbitrarily using the style from one of their arglists. You can find out which symbols are involved in collisions with:

```
(swank:print-indentation-lossage)
```

If a collision causes you irritation, don't have a nervous breakdown, just override the Elisp symbol's `common-lisp-indent-function` property to your taste. SLIME won't override your custom settings, it just tries to give you good defaults.

A more subtle issue is that imperfect caching is used for the sake of performance.[1] In an ideal world, Lisp would automatically scan every symbol for indentation changes after each command from Emacs. However, this is too expensive to do every time. Instead Lisp usually just scans the symbols whose home package matches the one used by the Emacs buffer where the request comes from. That is sufficient to pick up the indentation of most interactively-defined macros. To catch the rest we make a full scan of every symbol each time a new Lisp package is created between commands – that takes care of things like new systems being loaded.

You can use *M-x slime-update-indentation* to force all symbols to be scanned for indentation information.

[1] *Of course* we made sure it was actually too slow before making the ugly optimization.

3.4 Reader conditional fontification

SLIME automatically evaluates reader-conditional expressions in source buffers and "grays out" code that will be skipped for the current Lisp connection.

4 REPL: the "top level"

SLIME uses a custom Read-Eval-Print Loop (REPL, also known as a "top level"). The REPL user-interface is written in Emacs Lisp, which gives more Emacs-integration than the traditional `comint`-based Lisp interaction:

- Conditions signalled in REPL expressions are debugged with SLDB.
- Return values are distinguished from printed output by separate Emacs faces (colours).
- Emacs manages the REPL prompt with markers. This ensures that Lisp output is inserted in the right place, and doesn't get mixed up with user input.

4.1 REPL commands

RET `slime-repl-return`
Evaluate the current input in Lisp if it is complete. If incomplete, open a new line and indent. If a prefix argument is given then the input is evaluated without checking for completeness.

C-RET `slime-repl-closing-return`
Close any unmatched parenthesis and then evaluate the current input in Lisp. Also bound to *M-RET*.

C-j `slime-repl-newline-and-indent`
Open and indent a new line.

C-c C-c `slime-interrupt`
Interrupt the Lisp process with `SIGINT`.

TAB `slime-complete-symbol`
Complete the symbol at point.

C-c C-o `slime-repl-clear-output`
Remove the output and result of the previous expression from the buffer.

C-c C-t `slime-repl-clear-buffer`
Clear the entire buffer, leaving only a prompt.

4.2 Input navigation

C-a `slime-repl-bol`
Go to the beginning of the line, but stop at the REPL prompt.

M-n
M-p
M-s
M-r `slime-repl-{next,previous}-input`
`slime-repl-{next,previous}-matching-input`
`comint`-style input history commands.

C-c C-n
C-c C-p `slime-repl-next-prompt, slime-repl-previous-prompt`
Move between the current and previous prompts in the REPL buffer.

C-M-a
C-M-e `slime-repl-beginning-of-defun, slime-repl-end-of-defun` These commands are like `beginning-of-defun` and `end-of-defun`, but when used inside the REPL input area they instead go directly to the beginning or the end, respectively.

4.3 Shortcuts

"Shortcuts" are a special set of REPL commands that are invoked by name. To invoke a shortcut you first press `,` (comma) at the REPL prompt and then enter the shortcut's name when prompted.

Shortcuts deal with things like switching between directories and compiling and loading Lisp systems. The exact set of shortcuts is not currently documented in this manual, but you can use the `help` shortcut to list them interactively.

5 SLDB: the SLIME debugger

SLIME has a custom Emacs-based debugger called SLDB. Conditions signalled in the Lisp system invoke SLDB in Emacs by way of the Lisp `*DEBUGGER-HOOK*`.

SLDB pops up a buffer when a condition is signalled. The buffer displays a description of the condition, a list of restarts, and a backtrace. Commands are offered for invoking restarts, examining the backtrace, and poking around in stack frames.

5.1 Examining frames

Commands for examining the stack frame at point.

t `sldb-toggle-details`
: Toggle display of local variables and `CATCH` tags.

v `sldb-show-source`
: View the frame's current source expression. The expression is presented in the Lisp source file's buffer.

e `sldb-eval-in-frame`
: Evaluate an expression in the frame. The expression can refer to the available local variables in the frame.

d `sldb-pprint-eval-in-frame`
: Evaluate an expression in the frame and pretty-print the result in a temporary buffer.

D `sldb-disassemble`
: Disassemble the frame's function. Includes information such as the instruction pointer within the frame.

i `sldb-inspect-in-frame`
: Inspect the result of evaluating an expression in the frame.

5.2 Invoking restarts

a `sldb-abort`
: Invoke the `ABORT` restart.

q `sldb-quit`
: "Quit" – `THROW` to a tag that the top-level SLIME request-loop catches.

c `sldb-continue`
: Invoke the `CONTINUE` restart.

0 ... 9 Invoke a restart by number.

Restarts can also be invoked by pressing *RET* or *Mouse-2* on them in the buffer.

5.3 Navigating between frames

n

p `sldb-down, sldb-up`
Move between frames.

M-n

M-p `sldb-details-{down,up}`
Move between frames "with sugar": hide the details of the original frame and display the details and source code of the next. Sugared motion makes you see the details and source code for the current frame only.

5.4 Miscellaneous Commands

r `sldb-restart-frame`
Restart execution of the frame with the same arguments it was originally called with. (This command is not available in all implementations.)

R `sldb-return-from-frame`
Return from the frame with a value entered in the minibuffer. (This command is not available in all implementations.)

s `sldb-step`
Step to the next expression in the frame. (This command is not available in all implementations.)

B `sldb-break-with-default-debugger`
Exit SLDB and debug the condition using the Lisp system's default debugger.

: `slime-interactive-eval`
Evaluate an expression entered in the minibuffer.

6 Extras

6.1 slime-selector

The `slime-selector` command is for quickly switching to important buffers: the REPL, SLDB, the Lisp source you were just hacking, etc. Once invoked the command prompts for a single letter to specify which buffer it should display. Here are some of the options:

`?` A help buffer listing all `slime-selectors`'s available buffers.

`r` The REPL buffer for the current SLIME connection.

`d` The most recently activated SLDB buffer for the current connection.

`l` The most recently visited `lisp-mode` source buffer.

`s` The `*slime-scratch*` buffer. See [slime-scratch], page 6.

`slime-selector` doesn't have a key binding by default but we suggest that you assign it a global one. You can bind *C-c s* like this:

```
(global-set-key "\C-cs" 'slime-selector)
```

And then you can switch to the REPL from anywhere with *C-c s r*.

The macro `def-slime-selector-method` can be used to define new buffers for `slime-selector` to find.

6.2 slime-autodoc-mode

`slime-autodoc-mode` is an additional minor-mode for automatically showing information about symbols near the point. For function names the argument list is displayed and for global variables we show the value. This is a clone of `eldoc-mode` for Emacs Lisp.

The mode can be enabled in the `slime-setup` call of your `~/.emacs`:

```
(slime-setup :autodoc t)
```

6.3 Multiple connections

SLIME is able to connect to multiple Lisp processes at the same time. The *M-x slime* command, when invoked with a prefix argument, will offer to create an additional Lisp process if one is already running. This is often convenient, but it requires some understanding to make sure that your SLIME commands execute in the Lisp that you expect them to.

Some buffers are tied to specific Lisp processes. Each Lisp connection has its own REPL buffer, and all expressions entered or SLIME commands invoked in that buffer are sent to the associated connection. Other buffers created by SLIME are similarly tied to the connections they originate from, including SLDB buffers, apropos result listings, and so on. These buffers are the result of some interaction with a Lisp process, so commands in them always go back to that same process.

Commands executed in other places, such as `slime-mode` source buffers, always use the "default" connection. Usually this is the most recently established connection, but this can be reassigned via the "connection list" buffer:

C-c C-x c `slime-list-connections`
: Pop up a buffer listing the established connections.

The buffer displayed by `slime-list-connections` gives a one-line summary of each connection. The summary shows the connection's serial number, the name of the Lisp implementation, and other details of the Lisp process. The current "default" connection is indicated with an asterisk.

The commands available in the connection-list buffer are:

RET `slime-goto-connection`
: Pop to the REPL buffer of the connection at point.

d `slime-connection-list-make-default`
: Make the connection at point the "default" connection. It will then be used for commands in `slime-mode` source buffers.

g `slime-update-connection-list`
: Update the connection list in the buffer.

q `slime-temp-buffer-quit`
: Quit the connection list (kill buffer, restore window configuration).

6.4 Typeout frames

A "typeout frame" is a special Emacs frame which is used instead of the echo area (minibuffer) to display messages from SLIME commands. This is an optional feature. The advantage of a typeout frame over the echo area is that it can hold more text, it can be scrolled, and its contents don't disappear when you press a key. All potentially long messages are sent to the typeout frame, such as argument lists, macro expansions, and so on.

M-x slime-ensure-typeout-frame
: Ensure that a typeout frame exists, creating one if necessary.

If the typeout frame is closed then the echo area will be used again as usual.

To have a typeout frame created automatically at startup you can use the `slime-connected-hook`:

```
(add-hook 'slime-connected-hook 'slime-ensure-typeout-frame)
```

7 Customization

7.1 Emacs-side

The Emacs part of SLIME can be configured with the Emacs `customize` system, just use *`M-x customize-group slime RET`*. Because the customize system is self-describing, we only cover a few important or obscure configuration options here in the manual.

`slime-truncate-lines`
: The value to use for `truncate-lines` in line-by-line summary buffers popped up by SLIME. This is `t` by default, which ensures that lines do not wrap in backtraces, apropos listings, and so on. It can however cause information to spill off the screen.

`slime-multiprocessing`
: This should be set to `t` if you want to use multiprocessing (threads) in your Lisp system. It causes any necessary initialization to be performed during Lisp server startup.

`slime-complete-symbol-function`
: The function to use for completion of Lisp symbols. Three completion styles are available. The default `slime-complete-symbol*` performs completion "in parallel" over the hyphen-delimited sub-words of a symbol name.[1] Formally this means that "`a-b-c`" can complete to any symbol matching the regular expression "`^a.*-b.*-c.*`" (where "dot" matches anything but a hyphen). Examples give a more intuitive feeling:

  - `m-v-b` completes to `multiple-value-bind`.
  - `w-open` is ambiguous: it completes to either `with-open-file` or `with-open-stream`. The symbol is expanded to the longest common completion (`with-open-`) and the point is placed at the first point of ambiguity, which in this case is the end.
  - `w--stream` completes to `with-open-stream`.

  An alternative is `slime-simple-complete-symbol`, which completes in the usual Emacs way. Finally, there is `slime-fuzzy-complete-symbol`, which is quite different from both of the above and tries to find best matches to an abbreviated symbol. It also has its own keybinding, defaulting to *`C-c M-i`*. See [slime-fuzzy-complete-symbol], page 7, for more information.

`slime-translate-to-lisp-filename-function`
`slime-translate-from-lisp-filename-function`
: These functions can be used to translate filenames between Emacs and the Lisp system. They are useful if you run Emacs and Lisp on separate machines which share a common file system but use a different directory structure (different "mount points"). This is most common with SMB-based file sharing.

[1] This style of completion is modelled on '`completer.el`' by Chris McConnell. That package is bundled with ILISP.

`slime-net-coding-system`
: If you want to transmit Unicode characters between Emacs and the Lisp system, you should customize this variable. E.g., if you use SBCL, you can set:

```
(setq slime-net-coding-system 'utf-8-unix)
```

To actually display Unicode characters you also need appropriate fonts, otherwise the characters will be rendered as hollow boxes. If you are using Allegro CL and GNU Emacs, you can also use `emacs-mule-unix` as coding system. GNU Emacs has often nicer fonts for the latter encoding.

7.1.1 Hooks

`slime-mode-hook`
: This hook is run each time a buffer enters `slime-mode`. It is most useful for setting buffer-local configuration in your Lisp source buffers. An example use is to enable `slime-autodoc-mode` (See Section 6.2 [slime-autodoc-mode], page 16.)

`slime-connected-hook`
: This hook is run when SLIME establishes a connection to a Lisp server. An example use is to create a Typeout frame (See Section 6.4 [Typeout frames], page 17.)

`sldb-hook`
: This hook is run after SLDB is invoked. The hook functions are called from the SLDB buffer after it is initialized. An example use is to add `sldb-print-condition` to this hook, which makes all conditions debugged with SLDB be recorded in the REPL buffer.

7.2 Lisp-side (Swank)

The Lisp server side of SLIME (known as "Swank") offers several variables to configure. The initialization file '`~/.swank.lisp`' is automatically evaluated at startup and can be used to set these variables.

7.2.1 Communication style

The most important configurable is `SWANK:*COMMUNICATION-STYLE*`, which specifies the mechanism by which Lisp reads and processes protocol messages from Emacs. The choice of communication style has a global influence on SLIME's operation.

The available communication styles are:

`:FD-HANDLER`
: This style uses the classical Unix-style "`select()`-loop." Swank registers the communication socket with an event-dispatching framework (such as `SERVE-EVENT` in CMUCL and SBCL) and receives a callback when data is available. In this style requests from Emacs are only detected and processed when Lisp enters the event-loop. This style is simple and predictable.

`:SIGIO` This style uses *signal-driven I/O* with a `SIGIO` signal handler. Lisp receives requests from Emacs along with a signal, causing it to interrupt whatever it is doing to serve the request. This style has the advantage of responsiveness, since Emacs can perform operations in Lisp even while it is busy doing other things. It also allows Emacs to issue requests concurrently, e.g. to send one long-running request (like compilation) and then interrupt that with several short requests before it completes. The disadvantages are that it may conflict with other uses of `SIGIO` by Lisp code, and it may cause untold havoc by interrupting Lisp at an awkward moment.

`:SPAWN` This style uses multiprocessing support in the Lisp system to execute each request in a separate thread. This style has similar properties to `:SIGIO`, but it does not use signals and all requests issued by Emacs can be executed in parallel.

The default request handling style is chosen according to the capabilities your Lisp system. The general order of preference is `:SPAWN`, then `:SIGIO`, then `:FD-HANDLER`. You can check the default style by calling `SWANK-BACKEND:PREFERRED-COMMUNICATION-STYLE`. You can also override the default by setting `SWANK:*COMMUNICATION-STYLE*` in your Swank init file.

7.2.2 Other configurables

These Lisp variables can be configured via your '`~/.swank.lisp`' file:

`SWANK:*CONFIGURE-EMACS-INDENTATION*`
: This variable controls whether indentation styles for `&body`-arguments in macros are discovered and sent to Emacs. It is enabled by default.

`SWANK:*GLOBALLY-REDIRECT-IO*`
: When true this causes the standard streams (`*standard-output*`, etc) to be globally redirected to the REPL in Emacs. When `NIL` (the default) these streams are only temporarily redirected to Emacs using dynamic bindings while handling requests. Note that `*standard-input*` is currently never globally redirected into Emacs, because it can interact badly with the Lisp's native REPL by having it try to read from the Emacs one.

`SWANK:*GLOBAL-DEBUGGER*`
: When true (the default) this causes `*DEBUGGER-HOOK*` to be globally set to `SWANK:SWANK-DEBUGGER-HOOK` and thus for SLIME to handle all debugging in the Lisp image. This is for debugging multithreaded and callback-driven applications.

`SWANK:*SLDB-PRINTER-BINDINGS*`
`SWANK:*MACROEXPAND-PRINTER-BINDINGS*`
`SWANK:*SWANK-PPRINT-BINDINGS*`
: These variables can be used to customize the printer in various situations. The values of the variables are association lists of printer variable names with the corresponding value. E.g., to enable the pretty printer for formatting backtraces in SLDB, you can use:

```
(push '(*print-pretty* . t) swank:*sldb-printer-bindings*).
```

`SWANK:*USE-DEDICATED-OUTPUT-STREAM*`

This variable controls an optimization for sending printed output from Lisp to Emacs. When `t` (the default) a separate socket is established solely for Lisp to send printed output to Emacs through. Without the optimization it is necessary to send output in protocol-messages to Emacs which must then be decoded, and this doesn't always keep up if Lisp starts "spewing" copious output.

`SWANK:*LOG-EVENTS*`

Setting this variable to `t` causes all protocol messages exchanged with Emacs to be printed to `*TERMINAL-IO*`. This is useful for low-level debugging and for observing how SLIME works "on the wire." The output of `*TERMINAL-IO*` can be found in your Lisp system's own listener, usually in the buffer `*inferior-lisp*`.

8 Credits

The soppy ending...

Hackers of the good hack

SLIME is an Extension of SLIM by Eric Marsden. At the time of writing, the authors and code-contributors of SLIME are:

Helmut Eller	Luke Gorrie	Marco Baringer
Peter Seibel	Alan Ruttenberg	Daniel Barlow
Edi Weitz	Wolfgang Jenkner	Matthias Koeppe
Martin Simmons	Lawrence Mitchell	Christophe Rhodes
Brian Downing	Bill Clementson	Thomas Schilling
Michael Weber	Matthew Danish	James Bielman
Thomas F. Burdick	Lars Magne Ingebrigtsen	John Paul Wallington
Bryan O'Connor	Antonio Menezes Leitao	Alan Shutko
Utz-Uwe Haus	Tiago Maduro-Dias	Robert Lehr
Robert E. Brown	Raymond Toy	Jouni K Seppanen
Eric Blood	Eduardo Muoz	Chris Capel
Bjørn Nordbø	Andras Simon	Zach Beane
Wolfgang Mederle	Travis Cross	Thomas Burdick
Sean O'Rourke	Russell McManus	Rui Patrocnio
Reini Urban	Pawel Ostrowski	Nikodemus Siivola
Lynn Quam	Lasse Rasinen	Julian Stecklina
Juergen Gmeiner	Jan Rychter	James McIlree
Ivan Boldyrev	Ignas Mikalajunas	Ian Eslick
Hannu Koivisto	Frederic Brunel	Christian Lynbech
Brian Mastenbrook	Barry Fishman	Andreas Fuchs
Alexey Dejneka	Alan Caulkins	

... not counting the bundled code from '`hyperspec.el`', *CLOCC*, and the *CMU AI Repository*.

Many people on the `slime-devel` mailing list have made non-code contributions to SLIME. Life is hard though: you gotta send code to get your name in the manual. `:-)`

Thanks!

We're indebted to the good people of `common-lisp.net` for their hosting and help, and for rescuing us from "Sourceforge hell."

Implementors of the Lisps that we support have been a great help. We'd like to thank the CMUCL maintainers for their helpful answers, Craig Norvell and Kevin Layer at Franz providing Allegro CL licenses for SLIME development, and Peter Graves for his help to get SLIME running with ABCL.

Most of all we're happy to be working with the Lisp implementors who've joined in the SLIME development: Dan Barlow and Christophe Rhodes of SBCL, Gary Byers of OpenMCL,

and Martin Simmons of LispWorks. Thanks also to Alain Picard and Memetrics for funding Martin's initial work on the LispWorks backend!